UMIACS-TR-90-38 March 1990
CS-TR -2432

A TCP Instrumentation and Its Use in Evaluating Roundtrip-time Estimators*

Dheeraj Sanghi, M.C.V. Subramaniam, A. Udaya Shankar†,
Ólafur Gudmundsson, and Pankaj Jalote†

Systems Design and Analysis Group and
Department of Computer Science
University of Maryland
College Park, MD 20742

†Also with University of Maryland
Institute for Advanced Computer Studies

COMPUTER SCIENCE TECHNICAL REPORT SERIES

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

UMIACS-TR-90-38 March 1990
CS-TR -2432

A TCP Instrumentation and Its Use in Evaluating Roundtrip-time Estimators*

Dheeraj Sanghi, M.C.V. Subramaniam, A. Udaya Shankar†,
Ólafur Gudmundsson, and Pankaj Jalote†

Systems Design and Analysis Group and
Department of Computer Science
University of Maryland
College Park, MD 20742

†Also with University of Maryland
Institute for Advanced Computer Studies

Abstract

We describe an instrumentation of TCP/IP that monitors TCP connections and provides values of internal variables of the implementation. We define interface events for a TCP/IP connection, describe how traces are obtained, and how application processes initiate trace collection. The instrumentation has been implemented in 4.3BSD UNIX.‡ The instrumented TCP/IP provides a flexible environment for experimental studies. Using the instrumentation, we have studied the performance of different roundtrip-time estimators in the Internet environment. One conclusion of our study is that clock resolution is an important parameter, and the resolution currently used in UNIX implementations of TCP is woefully inadequate. Another conclusion is that, with an adequate clock resolution, a recently proposed estimator performs substantially better than the estimator suggested in the TCP specifications.

Keywords: TCP (Transmission Control Protocol), Information Transfer, Computer networks, (ICP)

*This research was supported in part by a grant from UNISYS Corp.,by the U. S. Army Strategic Defense Command under contract DSAG60-87-C-0066, and by the U.S. Navy (Office of Naval Research) under contract N00014-87-K-0241. The views, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official U. S. Army or Navy position, policy, or decision, unless so designated by other official documentation.
A. Udaya Shankar was also supported by the National Science Foundation grant NCR.80-04500.

‡UNIX is a trademark of AT&T Bell Laboratories.

Contents

1 Introduction

The Transmission Control Protocol (TCP) [14] is a connection-oriented, transport layer protocol that is used extensively in computer networks, both local-area and wide-area. TCP operates above the Internet Protocol (IP) [13], and provides reliable data transfer service to application protocols, such as file transfer and remote login.

IP provides TCP with virtual communication channels between every two host computers of the network. However, the virtual channels are unreliable, especially in a wide-area network, such as the Internet [5], where the channels are implemented by store-and-forward routing. They can lose, reorder and duplicate messages in transit. Furthermore, they display *congestive* behavior, by which we mean that their delay and loss characteristics depend significantly on the number of messages in transit in the channel. Typically, once this number exceeds a certain threshold, congestion sets in; message delays increase drastically and throughput levels off or decreases.

To achieve reliable data transfer over such virtual channels, TCP uses a sliding window mechanism, involving data sequence numbers, acknowledgement messages, send and receive windows, and retransmissions. Consider data transfer from a *source* application entity to a *destination* application entity. Let us refer to the TCP entity at the source (destination) as the source (destination) TCP entity.

The source application entity periodically produces data and passes it to the source TCP entity, which assigns increasing *sequence numbers* to successive data octets. The source TCP entity buffers the data octets until they are acknowledged by the destination TCP entity. The *send window* refers to the set of (contiguous) sequence numbers corresponding to the buffered data. Periodically, the source TCP entity sends *packets*, each containing one or more contiguous data octets accompanied by the sequence number of the first octet and the number of octets.

The destination TCP entity maintains a set of (contiguous) sequence numbers, referred to as the *receive window*. Data octets below the receive window have been passed to the destination application entity. Data octets received out of sequence but within the receive window are buffered. Periodically, the destination TCP entity sends an acknowledgement indicating the current receive window.

The source TCP entity maintains an *estimator* for the roundtrip time. When the source TCP entity sends a packet, it starts a *retransmission timer* with a timeout equal to the current value of the estimator. If the timer

expires and the packet is not yet acknowledged, the packet is retransmitted.

While the sliding window mechanism effectively ensures that data is not delivered out of sequence [18], obtaining good performance over congestive channels is an open research area that is becoming increasingly important as networks become larger and more heterogeneous [4, 7, 8, 11, 12, 20].

The performance of a TCP connection depends on various *policies* employed by the TCP entities regarding transmission, retransmission, round-trip time estimation, window sizes, etc. Due to the congestive nature of the channels, there is considerable interaction between the policies and the amount of congestion in the network. To put it another way, a TCP implementation with bad policies, not only offers low performance to its application entities, but can also severely degrade the overall performance of the network by introducing congestion.

To understand the behavior of such a complex system, it is essential to do experimental work with instrumented TCP/IP implementation. Recently, there has been much effort in this direction [3, 4, 6, 7, 12, 17]. Cabrera et al [3] have studied TCP connections across two Ethernets connected by a VAX gateway. They examine throughput versus TCP packet size. Van Jacobson [6, 7] has studied TCP connections across two 10Mbs Ethernets connected by a succession of IP level links, including a bottleneck link of 230 Kbs. He has implemented algorithms for roundtrip-time variance estimation, exponential retransmission backoff, and slow start. Clark [4] has studied connections across Ethernets connected by gateways, and has implemented policies that reduce congestion. Nagle [12] has done similar work over local and wide-area connections. Seo et al [17] have studied the performance of SATNET, which links the Internet in North America to European networks. SATNET itself consists of four nodes fully interconnected by two multi-access 64 Kbs satellite channels with a propagation delay of 0.8 seconds.

There are two facilities available in UNIX for studying network behavior. One is the TCP trace facility, which works by setting the SO_DEBUG option on BSD sockets. This is useful for debugging connections, but not for gathering performance data, because it uses the kernel printf routine to print state and packet information while processing a packet. The kernel printf routine is not interrupt driven, and all system activities are suspended, while it is executing. This can skew the observations considerably. The other facility is the tcpdump program, which is used for passive monitoring from a host on the same local network as the test host. This facility does not affect the test host, but cannot access internal parameters of TCP, such as the send window.

Our goal was to obtain a general instrumentation of TCP/IP that would allow us to study transient and steady-state correlations between different parameters of interest, in both local-area and Internet environments.

Our Instrumentation

In this paper, we discuss an instrumentation of TCP/IP, that has been implemented in 4.3BSD UNIX[2] and is currently running on a VAXstation 3200[3]. Given a TCP connection between two applications, say a client and a server, our instrumented TCP/IP logs an entry for every packet that crosses an interface. Each log entry contains the following information: the time of occurrence as indicated by a local clock, values of different fields on the packet, and current values of identified state variables of the connection. Log entries can be recorded either at the client host, or at the server host, or at both hosts. In each host, the log entries are collected in a *trace*. Logging can be initiated by either the client or the server, or by both. In the case of logging at both client and server hosts, one option is to include a unique *transmission number* in each TCP packet sent. This allows identification of lost and duplicate packets.

An extremely powerful use of this instrumentation is to have both the client and the server on the same host, with the packets being routed via one or more specified gateways. In this case, there is a single trace for both ends of the connection. From this trace, we can obtain parameters such as one-way delay of each packet, number of packets in transit, number of packets lost, etc., and study the evolution of these parameters with time and their cross-correlations. This capability of the instrumentation appears to be unique.

Having both client and server on the same host has other advantages. It avoids the need for synchronizing the clocks in two hosts. It allows us to experiment with multi-gateway channels in the Internet with only a single host running the instrumented kernel.

We have also developed a set of post processing tools to analyze the trace and present results in statistical and graphical forms. With these tools our system provides an excellent environment for performing experimental studies. Due to the detailed information available about the behavior, this instrumentation can be used to validate analytic models of protocol behavior (such as in [1, 9]), which often state the dynamic properties of different

[2] A preliminary version was also implemented in SUN OS 3.2.

[3] VAXstation is a trademark of Digital Equipment Corporation.

variables.

Evaluation of roundtrip-time estimators

The *roundtrip time* of a packet is the time interval between sending the packet and receiving its acknowledgement. In TCP, the roundtrip times observed by a TCP entity are the *only* information that it has concerning the amount of congestion currently in the network. It uses these roundtrip times to maintain an *estimate* of the current roundtrip time. When a packet is sent, this estimate is used to set the retransmission timeout of the packet.

Clearly, a good roundtrip-time estimator is essential for good TCP performance. If the estimate is too high, packet losses will be detected late. As a result, retransmissions will be delayed and throughput will decrease. If the estimate is too low, the TCP entity will retransmit packets that are still in transit. This may lead to congestion [12].

We have used our instrumented TCP/IP to evaluate the performance of different estimators. In this report, we investigate the effect of the clock resolution used to measure the roundtrip times. We also compare the roundtrip-time estimator suggested by Van Jacobson [7] against the one suggested in the TCP specification [14]. The *error* for a packet is defined as the difference between the value of the estimator at the time of sending the packet and the roundtrip time experienced by the packet. The sample standard deviation of these errors is the metric we use to evaluate the estimator.

Organization of the rest of the paper

In Section 2, we discuss the design issues involved in instrumenting a TCP/IP implementation. In Section 3, we discuss the UNIX implementation of the instrumentation. In Section 4, we discuss some experiments. In Section 5, we conclude and suggest future extensions of this work.

2 Instrumentation of TCP/IP

Figure 1 illustrates the protocol organization between two hosts A and B connected via the TCP/IP protocol. APP_A, TCP_A and IP_A are the Application, TCP, and IP entities in host A, respectively. The entities in host B are organized similarly. These entities define three *interfaces*, namely, APP/TCP, TCP/IP, and IP/Network. Packets can cross an interface in either direction. The natural time to collect information is when a packet crosses an interface.

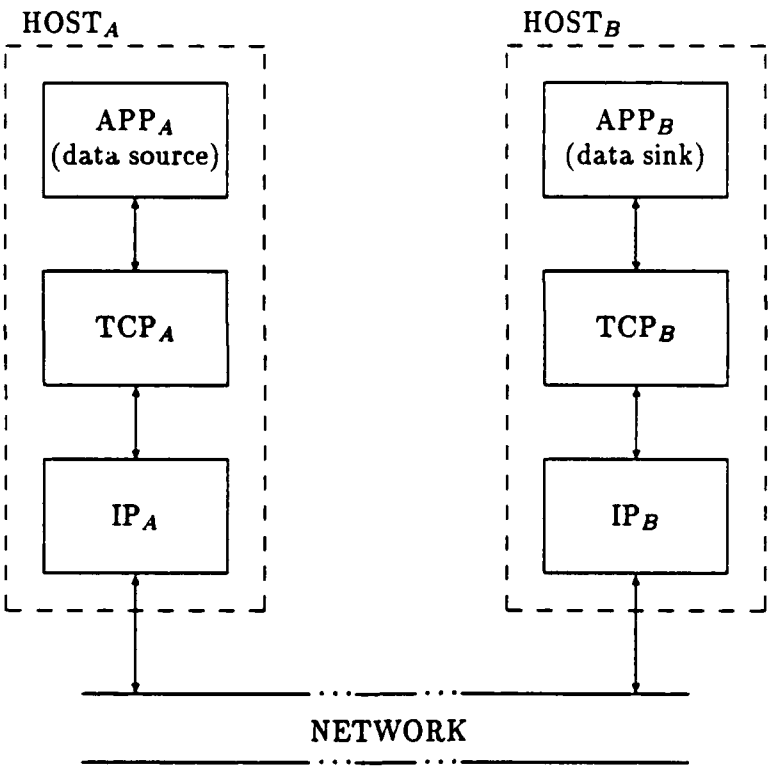


Figure 1: Organization of a TCP connection

2.1 Data Logging

Most application entities communicate according to the *client-server* model. In this model, an application entity is either a server or a client. Servers

provide a service (e.g. file transfer) to clients. Only clients can initiate requests for service.

An application entity can request either local logging or two-host logging. In *local logging*, only packet crossings on local host interfaces are logged. In *two-host logging*, in addition to logging at the local host, the remote host is requested to start logging at its interfaces. This request can be conveyed by sending a *transmission number* in the TCP option list.

Successive TCP packets (including retransmissions) have consecutively increasing transmission numbers, starting with 1. The transmission number is sent only if at least one of the applications has requested two-host logging. On receiving a packet with a transmission number in it, the TCP entity starts logging for that connection and begins to include transmission numbers in outgoing packets.

A special case of two-host logging is to have both the client and the server on the same host, with the packets of the connection being routed via one or more specified gateways (using the IP LSRR option [10]).

2.2 Format of a Log Entry

A log entry is made when a packet crosses an interface. Every log entry contains a *timestamp* obtained from a clock in the host, source and destination port numbers, and the transmission number. Additional fields in the log entry depend on the interface at which it is logged and are described below:

Application/TCP interface:

- Number of outstanding octets (i.e. number of octets given by the application that have not been acknowledged)

TCP/IP interface:

- Fields from the packet:

  send sequence number

  acknowledgement sequence number

  receive window size

  packet size

  packet header size

  TCP header flags

send window size

- Outstanding data in the connection at that host.

IP/Network interface:

- Fields from the packet

  IP time to live

  IP header length

  IP packet length

The trace of a connection contains (arguably) all the information needed for analysis. From it, we can extract the values of state variables at different instants, study relationships between them, and obtain performance measures.

For example, a packet is considered lost, if there is a log record indicating it was sent but none indicating that it has been received. The number of times an octet has been retransmitted can be obtained by scanning the log records of send events. The throughput of a connection is the number of octets sent, divided by the total time of the connection.

2.3 Implementation Issues

A major requirement of the instrumentation is that it should have minimal effect on the results.

A log entry is appended to the trace every time a packet crosses an interface between the two entities. To minimize the effect of logging, the log entry for a packet is made after the packet has been sent.

Because the number of packets sent in a connection can be large, the size of the trace can exceed the size of physical memory. However, we cannot allow the TCP or IP entities to append log entries to a disk file, because that would be very slow, thereby affecting the experiment. Our choice was to append the log entries to a buffer in physical memory. A *reader* process periodically transfers these entries to a disk file.

Access to the shared memory by the TCP entity and the reader process has to be mutually exclusive. We try to keep the critical section access to minimum. Our method is to have a linked list of buffers, with the critical section involving only the modification of pointers. The reading and writing of the buffers is done outside the critical section. If there is no empty buffer

available, TCP and IP do not make a log entry. This avoids blocking when the reader process is slow.[4]

The logging of a connection should not affect other connections that have not opted for logging. In our implementation, we set a flag for each connection for which logging is desired. No logging is done if this flag is not set.

[4] Also, the user may not have started the reader process.

3 Implementation under UNIX

In 4.3BSD UNIX, the TCP/IP routines are part of the kernel. Here we describe briefly the modifications that we have made to the kernel. A detailed description can be found in [15].

The TCP and IP entities write their log entries in main memory. For this purpose, a kernel memory area that is accessible to the TCP and IP routines is required. The tcp_init() routine, which is executed as a part of kernel initialization procedure, has been modified to allocate a block of memory. This block is organized into two linked lists of records – the *empty* list and the *full* list. Each record can hold one log entry. Initially, all the records are in the empty list.

When a packet crosses an interface, the modified TCP and IP routines write a log entry in an empty record, and append it to the full list. If there is no record in the empty list, no log entry is appended.

There is a *reader* process that reads log entries from the memory and writes them to a disk file. The reader process views the memory as a read-only device called *netlog*. A device driver has been written for this pseudo-device.

The reader process is started at the beginning of the experiment and runs throughout the experiment. It employs blocking I/O so that it is suspended when there are no records in the full list. It is woken up by the TCP and IP entities when they append a log entry to the full list. The TCP/IP entities and the netlog device driver ensure that accesses to the free and empty lists are mutually exclusive by raising the priority of the cpu.

The traces of all the connections are written in the same file. The trace for a particular connection can be extracted during post processing.

3.1 Application Interface

An application entity performs different activities to establish a connection, depending on whether it is a client or a server [10]. A server entity executes the following steps:

S_1: Inform the local TCP entity of its willingness to provide service by creating a *socket*.

S_2: Inform the TCP entity that it is ready to receive service requests.

S_3: Wait for an incoming connection request from a client entity.

S_4: Service the connection until termination.

A client entity executes the following steps:

C_1: Inform the local TCP entity of its need to get service. A socket is created for the client.

C_2: Request connection to the server.

C_3: Once the connection is established the client may begin requesting service.

UNIX provides the setsockopt() call for applications to set different socket options. We have modified the setsockopt() call such that the logging option can also be set by an application entity.

For each connection, UNIX maintains a number of data structures to support inter-process communication. Here, we mention the ones relevant to our discussion. For each connection in the system, three structures, called *tcpcb*, *inpcb*, and *socket* are maintained. Tcpcb contains the values of TCP state variables. Inpcb contains the protocol independent information like routing entry and the IP options. Socket has pointers to send and receive buffer queues. These structures have pointers to each other. The inpcbs of all the TCP connections in the system are linked in a list. We keep the transmission number for a connection in a separate *mbuf* (the unit of memory buffer in the UNIX kernel), which is accessed through a pointer in tcpcb. Recall that the transmission number is used to uniquely identify packets (see 'Our Instrumentation' subsection in Section 1).

3.2 Modifications to TCP/IP Routines

The TCP/IP routines have been modified to append log entries to the kernel memory area. In our current implementation, we have instrumented the TCP/IP and the IP/Network interfaces. Here we briefly describe the modifications that have been made to the TCP/IP routines.

Packet from TCP to IP: The tcp_output() routine takes the data to be sent from the socket queues. It appends the TCP header to the data and passes the packet to IP through a call to ip_output(). The tcp_output() routine has been modified to append a log entry at this stage. The timestamp is obtained just before the call is made. The log entry is appended after the call returns, thereby avoiding a delay (due to logging) in sending the packet.

Packet from IP to Network: IP receives a packet from TCP through the ip_output() routine. This routine has been modified to append a log

entry just before it makes a call to the network interface driver. It decides whether or not to log by scanning the flags passed to it by tcp_output().

Packet from Network to IP: The routine that handles incoming packets for IP is ip_intr(). It removes the packet from the queue and determines whether the packet is destined for the local host or is to be forwarded to another host. In the former case, it passes the packet to the upper layer protocol. The ip_intr() routine has been modified to append a log entry just before calling the upper layer protocol. The time stamp for this entry is taken in the beginning of the routine. To decide whether the connection to which the packet belongs has the logging option set, the tcpcb of this connection is examined.

Packet from IP to TCP: The tcp_input() routine processes an incoming packet for TCP. It calls in_pcblookup() to determine which connection the packet should go to. The tcp_input() routine has been modified to append a log entry if that connection has the logging option set.

Two special cases arise at this stage.

(a) When a SYN packet is received for a socket, TCP creates new instances of the socket, the inpcb, and the tcpcb data structures for the new connection. This portion of tcp_input() has been modified to determine whether the parent socket had the logging option set. If it did, then tcp_input() sets the option for the new socket as well.

(b) If a packet is received with the transmission number in the TCP options, the modified tcp_input() routine sets the two-host logging option for the connection.

The Transmission Number: Conventional TCP uses only one option, TCP_MAXSEG, indicating the maximum segment size. This is sent along with the SYN packets that the two hosts exchange while establishing a connection. We have introduced another option called TCP_TRNUM for the transmission number. This option is sent on every packet of a connection that has two-host logging option set. The tcp_output() routine has been modified to send the TCP_TRNUM option.

The tcp_dooptions() routine processes the options in an incoming TCP packet. This routine has been modified to recognize the TCP_TRNUM option. If a transmission number is present and the two-host logging option has not already been set for the connection, then this routine sets the option.

4 Evaluating Roundtrip time Estimators

We have performed a number of experiments using our instrumented TCP/-IP. In this Section, we present some results to demonstrate the capabilities of our instrumentation and to compare different roundtrip-time estimators.

The TCP implementation in 4.3 BSD UNIX maintains several variables for setting the retransmission timeout of a packet, namely: SRTT, RTTVAR, RXT, Roundtrip_Timer, and Retransmission_Timer. SRTT is the "smoothed" average of measured roundtrip times. RTTVAR is the "smoothed" variance of measured roundtrip times. RXT is the current retransmission-timeout estimate. Roundtrip_Timer is used to measure roundtrip time of one packet. Retransmission_Timer is used to indicate when to retransmit.

When a packet is transmitted for the first time (i.e., contains no octet that has been transmitted already) and Roundtrip_Timer is not active, TCP records the sequence number of the first byte of the packet and starts the timer. Every 500 ms, a software clock interrupt increments Roundtrip_Timer by 1.[5] When an acknowledgement is received for that packet, The roundtrip time, denoted RTT, for that packet equals the value of Roundtrip_Timer multiplied by 500 ms. If the packet is retransmitted before its acknowledgement is received, the roundtrip-time measurement is aborted.

Each time an RTT is obtained, three of the above variables are updated as follows (this update scheme was introduced by Van Jacobson [7] and differs from the suggested in the TCP specification [14]):

$$\text{SRTT}_{new} = \alpha\ \text{SRTT} + (1-\alpha)\ \text{RTT}$$
$$\text{RTTVAR}_{new} = \alpha'\ \text{RTTVAR} + (1-\alpha')(|\ \text{RTT} - \text{SRTT}\ | - \text{RTTVAR})$$
$$\text{RXT}_{new} = \text{SRTT}_{new} + 2\ \text{RTTVAR}_{new}$$

TCP uses the values $\alpha = 7/8$ and $\alpha' = 6/8$.

When a packet is sent and Retransmission_Timer is not active, TCP sets it to the current value of RXT. Every 500 ms, a software clock interrupt (the same one that increments the active roundtrip timers) decrements the active retransmission timers of all TCP connections on that host. If the packet is not acknowledged before its Retransmission_Timer becomes zero, the packet is retransmitted and the timer is set with a value equal to *twice* the previous timeout value. If the packet is acknowledged before the timer becomes zero,

[5] Actually, it increments the active roundtrip timers of all TCP connections on that host.

the timer is reset to the current value of RXT if and only if there is still some outstanding packet.

From a trace, we can compute the roundtrip time of each packet. Using these, we can simulate the effect of different RXT functions. There is an assumption underlying our treatment; namely, that the roundtrip times experienced by the packets would remain the same. In reality, a different RXT function can cause packet transmission times to be different from those in the trace. This in turn can affect the network congestion and therefore the roundtrip times of the packets. Our assumption corresponds to ignoring this feedback effect. Certainly our assumption would be valid in situations of low user load.

We now identify the packets whose roundtrip times are used in simulating the TCP RXT functions. First, we point out that TCP only uses the roundtrip times of packets that were not retransmitted[6]. Thus, let $p_1, \ldots, p_N$ be the sequence of such packets sent in the connection. From the trace, we can obtain the transmission time, s_i, and the acknowledgement time, a_i, for each p_i. We have $\mathrm{RTT}_i = a_i - s_i$. Second, recall that a TCP entity uses only one retransmission timer and one roundtrip timer. This means that only the RTT_i's of non-overlapping packets are used in simulating an RXT, where p_i overlaps with p_j if and only if $s_i < s_j < a_i$.

Finally, we define the metrics used in evaluating an RXT function.

- Mean Square Error

  $$\mathrm{MSE} = \sqrt{\frac{\sum_i (e_i)^2}{N}},$$

  where $e_i = \mathrm{RXT}_i - \mathrm{RTT}_i$ and RXT_i is the retransmission-timeout estimate at the time packet p_i is sent.

- Mean Square Error of the Under-estimations

  $$\mathrm{MSE}^- = \sqrt{\frac{\sum_i (e_i)^2}{N}},$$

  where i ranges over the packet numbers for which $e_i < 0$. (Packet numbers are same as transmission numbers defined earlier.)

- Mean Square Error of the Over-estimations

  $$\mathrm{MSE}^+ = \sqrt{\frac{\sum_i (e_i)^2}{N}},$$

  where i ranges over the packet numbers for which $e_i > 0$.

[6] A packet is considered retransmitted if even one octet in this packet is retransmitted.

MSE, MSE^-, and MSE^+ indicate how close the roundtrip-time estimates are to the actual roundtrip times of packets. A high value of MSE^- implies a large number of unnecessary retransmissions. A high value of MSE^+ implies large delays in retransmissions of lost packets, resulting in under-utilization of the network.

Experiments

In each experiment that we describe here, there were two application processes, a data source and a data sink. (See Figure 1.) Both processes were on the host huginn.cs.umd.edu (which is a VAXstation 3200) at the Computer Science Department at Maryland. All packets and acknowledgements were routed via ucbvax.berkeley.edu at the University of California, Berkeley.

In experiment 1, the source generated 1 octet of data every second for 1000 seconds (for a total of 1000 octets). This experiment was carried out at night when the network load is typically low.

In experiment 2, the source generated 1000 octets of data 1000 times (for a total of 10^6 octets). The data was generated as fast as the local TCP entity could accept. This experiment was done during the day when the network load is typically higher. For timestamping the log records, we used the UNIX internal clock, with a resolution of 10 ms.

Experiment 1: Round trip times using 500 ms resolution

Figure 2 shows the RTTs, SRTT and RXT in experiment 1. The x-coordinate is the packet number. Each dot (.) represents an RTT measurement. Each asterisk (*) represents a packet lost in transit. 8 packets were lost in transit. The values of SRTT and RXT were calculated assuming the 500ms clock resolution used conventionally by TCP for RTT measurements.

Note that there is only one packet (number 683) whose RTT (1200 ms) exceeds the RXT value (1000 ms) at the time of its transmission. We notice from Figure 2 that TCP greatly overestimates the roundtrip time. The values of MSE, MSE^- and MSE^+ are 465, 6 and 465 respectively (also shown in Table 1).

Experiment 1: Round trip times assuming 10 ms resolution

We want to study the effect of increasing the clock resolution on the TCP roundtrip-time measurements. Figure 3 shows the RTTs of experiment 1, and the values of SRTT and RXT assuming a 10 ms clock resolution for RTT measurements. Note that our assumption that there is no feedback effect is valid in this experiment because the packets are spaced 1 second apart. Therefore, there is no interference between two successive packets.

The values of MSE, MSE^- and MSE^+ are 102, 26 and 98 respectively. It is clear from the Figures 2 and 3 and the values of MSE^+ in Table 1 that RXT values are much closer to the RTTs if a 10 ms clock resolution is used.

clock res.	MSE (-, +)
500 ms	465 (6, 465)
10 ms	102 (26, 98)

Table 1: Experiment 1 with different clock resolutions

Experiment 2: Round trip times using 500 and 10 ms resolution

Figure 4 shows the RTTs for experiment 2, and values of SRTT and RXT assuming a 500 ms clock resolution for RTT measurements. 15 packets were lost in transit in this experiment. Figure 5 shows the RTTs for experiment 2, and values of SRTT and RXT assuming a 10 ms clock resolution for RTT measurements. Here, our assumption of ignoring the feedback effect may not be valid.

The error metrics for these simulation are given in Table 2. We again notice that an increased resolution of the clock results in RXT values to be much closer to the RTTs.

clock res.	MSE (-, +)
500 ms	785 (0, 785)
10 ms	136 (28, 134)

Table 2: Experiment 2 with different clock resolutions

Changing estimator parameters

The value of α controls how rapidly SRTT adjusts to changing network conditions. A smaller value of α allows SRTT to adapt more swiftly. We next simulate the TCP RXT estimator with different values of α, with $\alpha' = \alpha$, and using both 500 ms and 10 ms clock resolutions. Table 3 shows the values of the error metrics for different values of α for experiment 1. Figure 6 shows these values graphically. Table 4 shows the values of error metrics for different values of α for experiment 2. Figure 7 shows these values graphically.

We observe that with clock resolution of 10 ms, MSE and MSE^+ remain approximately the same for different values of α. With clock resolution of 500 ms, MSE and MSE^+ decrease as the value of α decreases.

	$\alpha = \frac{7}{8}$	$\alpha = \frac{6}{8}$	$\alpha = \frac{5}{8}$	$\alpha = \frac{4}{8}$
clock res.	MSE (-, +)	MSE (-, +)	MSE (-, +)	MSE (-, +)
500 ms	463 (6, 463)	363 (32, 361)	362 (32, 360)	84 (72, 44)
10 ms	102 (25, 99)	101 (27, 97)	106 (28, 102)	111 (30, 107)

Table 3: Experiment 1 with different values of α ($\alpha' = \alpha$)

	$\alpha = \frac{7}{8}$	$\alpha = \frac{6}{8}$	$\alpha = \frac{5}{8}$	$\alpha = \frac{4}{8}$
clock res.	MSE (-, +)	MSE (-, +)	MSE (-, +)	MSE (-, +)
500 ms	840 (0, 840)	509 (1, 509)	425 (1, 425)	258 (140, 217)
10 ms	139 (26, 137)	136 (29, 133)	140 (31, 136)	142 (32, 138)

Table 4: Experiment 2 with different values of α ($\alpha' = \alpha$)

Increasing the number of packets whose RTT is measured

Recall that TCP does not measure roundtrip times of overlapping packets. We now simulate the TCP RXT estimator assuming TCP measures roundtrip times of all packets that are not retransmitted and whose acknowledgements are not lost.

Figure 8 shows the values of SRTT and RXT for experiment 1 under this assumption (along with observed RTTs). The error metrics are given in Table 5. We see that there is no difference between Tables 5 and 3, which is to be expected for lightly loaded conditions. Figures 9 and 10 show the values of SRTT and RXT for experiment 2. The error metrics are given in Table 6. Comparing Tables 6 and 4, we observe that no significant improvement is achieved by measuring the RTTs of more packets.

	$\alpha = \frac{7}{8}$	$\alpha = \frac{6}{8}$	$\alpha = \frac{5}{8}$	$\alpha = \frac{4}{8}$
clock res.	MSE (-, +)	MSE (-, +)	MSE (-, +)	MSE (-, +)
500 ms	463 (6, 463)	363 (32, 361)	362 (32, 360)	84 (72, 44)
10 ms	102 (25, 99)	101 (27, 97)	106 (28, 102)	111 (30, 107)

Table 5: Experiment 1 with RTTs measured of all possible packets

RXT estimator suggested in the TCP specification

The TCP specification [14] suggests that the retransmission timeout be calculated as

RXT = 2 SRTT

	$\alpha = \frac{7}{8}$	$\alpha = \frac{6}{8}$	$\alpha = \frac{5}{8}$	$\alpha = \frac{4}{8}$
clock res.	MSE (-, +)	MSE (-, +)	MSE (-, +)	MSE (-, +)
500 ms	882 (0, 882)	452 (1, 452)	389 (0, 389)	251 (141, 207)
10 ms	148 (23, 147)	140 (28, 138)	141 (31, 138)	143 (33, 139)

Table 6: Experiment 2 with RTTs measured of all possible packets

We next give the error metrics assuming this estimator, for different values of α, with $\alpha' = \alpha$. Table 7 gives the values for experiment 1 when roundtrip time is measured only for non-overlapping packets. Table 8 gives the values for experiment 1 when roundtrip time is measured for all possible packets. Table 9 gives the values for experiment 2 when roundtrip time is measured only for non-overlapping packets. Table 10 gives the values for experiment 2 when roundtrip time is measured for all possible packets.

We notice that RXT is considerably higher than RTTs, irrespective of the resolution of the clock measuring the RTTs, and of the number of packets whose roundtrip time is measured. When we compare these Tables with Tables 3-6, we see that this estimator is worse than Van Jacobson's estimator [7], which is currently used in UNIX.

	$\alpha = \frac{7}{8}$	$\alpha = \frac{6}{8}$	$\alpha = \frac{5}{8}$	$\alpha = \frac{4}{8}$
clock res.	MSE (-, +)	MSE (-, +)	MSE (-, +)	MSE (-, +)
500 ms	599 (0, 599)	686 (0, 686)	629 (0, 629)	544 (2, 544)
10 ms	555 (0, 555)	550 (0, 550)	550 (1, 550)	549 (1, 549)

Table 7: Expt. 1 with RXT = 2 SRTT and RTTs of non-overlapping packets

	$\alpha = \frac{7}{8}$	$\alpha = \frac{6}{8}$	$\alpha = \frac{5}{8}$	$\alpha = \frac{4}{8}$
clock res.	MSE (-, +)	MSE (-, +)	MSE (-, +)	MSE (-, +)
500 ms	599 (0, 599)	686 (0, 686)	629 (0, 629)	544 (2, 544)
10 ms	555 (0, 555)	550 (0, 550)	550 (1, 550)	549 (1, 549)

Table 8: Expt. 1 with RXT = 2 SRTT and RTTs of all possible packets

	$\alpha = \frac{7}{8}$	$\alpha = \frac{6}{8}$	$\alpha = \frac{5}{8}$	$\alpha = \frac{4}{8}$
clock	MSE (-, +)	MSE (-, +)	MSE (-, +)	MSE (-, +)
500 ms	1109 (0, 1109)	800 (0, 800)	725 (0, 725)	627 (0, 627)
10 ms	673 (0, 673)	674 (0, 674)	675 (0, 675)	676 (0, 676)

Table 9: Expt. 2 with RXT = 2 SRTT and RTTs of non-overlapping packets

	$\alpha = \frac{7}{8}$	$\alpha = \frac{6}{8}$	$\alpha = \frac{5}{8}$	$\alpha = \frac{4}{8}$
clock	MSE (-, +)	MSE (-, +)	MSE (-, +)	MSE (-, +)
500 ms	1224 (0, 1224)	824 (0, 824)	748 (0, 748)	662 (0, 662)
10 ms	700 (0, 700)	697 (0, 697)	697 (0, 697)	695 (0, 695)

Table 10: Expt. 2 with RXT = 2 SRTT and RTTs of all possible packets

5 Conclusion

In this report, we have described an instrumentation that can monitor selected TCP connections. The instrumentation scheme is designed to collect information at different interfaces in a TCP/IP implementation. The current version is implemented in 4.3BSD UNIX.

The instrumentation provides information about various performance measures and internal variables of an implementation. This can be useful in better understanding the working of the implementation, which in turn can help in determining optimal policies for TCP.

We have used the instrumentation to study the effect of different round-trip time estimators. From the results presented in this paper, it is clear that a high resolution clock is essential to obtain good estimates. It also appears that the RXT estimator suggested by Van Jacobson [7] performs better than the one suggested in the TCP specification [14].

Elsewhere [16], we have used our instrumentation to find the number of retransmissions, packets in transit, loss rate, etc, to study response time versus packet size, and to validate analytic models [1]. We believe that the instrumentation described here can be done on any communication protocol, to test the protocol, to measure its performance, and to validate analytic models. The statistics provided by such instrumentation would be a good reference point to compare TCP to other transport protocols. We are planning to instrument ISO protocols in the next version of BSD UNIX.

In the future, one can think of "log servers," just like file servers or remote login servers. A log server would allow a remote client to establish a connection, send or receive data according to a specified traffic pattern, generate a local trace, and ship the trace over to the client at the end of the experiment. This would allow TCP entities that do not have instrumentation to evaluate the performance of their policies.

Acknowledgement: We wish to thank Steve Miller of the University of Maryland Institute for Advanced Computer Studies (UMIACS) for explaining the internals of the UNIX interprocess communication mechanisms and helping us throughout the implementation work.

References

[1] Bolot, J., Shankar, A.U., Plateau, B.D., "Performance Analysis of Transport Protocols over Congestive Channels," Technical Report CS-TR-2004, UMIACS-TR-88-22, Department of Computer Science, University of Maryland, College Park, March 1988. Also to appear in *Performance Evaluation.*

[2] Bolot, J., Shankar, A.U., "Dynamical behavior of rate-based flow control mechanism," Technical Report CS-TR-2279.1, UMIACS-TR-89-67.1, Department of Computer Science, University of Maryland, College Park, October 1989. Also to appear in *Computer Communication Review.*

[3] Cabrera, L.F., Hunter, E., Karels, M.J., Mosher, D.A., "User-process Communication Performance in Networks of Computers," *IEEE Transaction of Software Engineering,* Vol. 14, No. 1, pp. 38-53, January 1988.

[4] Clark, D.D., "Window and Acknowledgement Strategy in TCP," RFC 813, Network Information Center (NIC), SRI International, Menlo Park, CA, July, 1982.

[5] Comer, D., *Internetworking with TCP/IP: principles, protocols and architectures,* Prentice Hall, 1988.

[6] Jacobson, V., "Maximum Ethernet Throughput," In the *Proceedings of the Ninth Internet Engineering Task Force,* The MITRE Corporation, McLean, VA, March 1988. Also available as posting to the tcp-ip electronic bulletin board, March 1988.

[7] Jacobson, V., "Congestion Avoidance and Control," *Proceedings of ACM SIGCOMM '88,* Stanford, California, pp. 314-329, August 1988.

[8] Karn, P., Partridge, C., "Improving round-trip time estimates in reliable transport protocols," *Proceedings of ACM SIGCOMM '87,* Stowe, Vermont, pp. 2-7, 1987.

[9] Lam, S.S., Hsieh, C., "Modeling Analysis and Optimal Routing of Flow-Controlled Communication Networks," *Proceedings of ACM SIGCOMM '87,* Stowe, Vermont, pp. 162-172, August 1987.

[10] Leffler, S.J., Joy, W.N., Fabry, R.S., Lapsley, P., Miller, S., Torek, C., *An Advanced 4.3BSD Interprocess Communication Tutorial,* Department of Electrical Engineering and Computer Science, University of California, Berkeley, 1986.

[11] Mills, D.L., Braun, H.W., "The NSFNET backbone network," *Proceedings of ACM SIGCOMM '87,* Stowe, Vermont, pp. 191-196, 1987.

[12] Nagle, J., "Congestion Control in IP/TCP Internetworks," RFC 896, Network Information Center (NIC), SRI International, Menlo Park, CA, January 1984.

[13] Postel, J. (editor), "Internet Protocol," RFC 791, Information Sciences Institute, University of Southern California, September 1981.

[14] Postel, J. (editor), "Transmission Control Protocol," RFC 793, Information Sciences Institute, University of Southern California, September 1981.

[15] Sanghi, D., Subramaniam, M.C.V., Gudmundsson, O., Caballero, M., Shankar, A.U., "Performance Instrumentation of TCP," Technical Report CS-TR-2009, UMIACS-TR-88-24, Department of Computer Science, University of Maryland, College Park, April 1988.

[16] Sanghi, D., Subramaniam, M.C.V., Shankar, A.U., Gudmundsson, O., Jalote, P., "Instrumenting a TCP Implementation," Technical Report CS-TR-2061, UMIACS-TR-88-50, Department of Computer Science, University of Maryland, College Park, July 1988.

[17] Seo, K., Crowcroft, J., Spilling, P., Laws, J., Leddy, J., "Distributed Testing and Measurement across the Atlantic Packet Satellite Network (SATNET)," *Proceedings of ACM SIGCOMM '88,* Stanford, California, pp. 235-246, August 1988.

[18] Shankar, A.U., "Verified data transfer protocols with variable flow control," *ACM Trans. Comput. Sys.,* Vol. 7, No. 3, pp. 281-316, August 1989.

[19] *UNIX Programmer's Manual (5),* 4.2 Berkeley Software Distribution, University of California, Berkeley, March 1988.

[20] Zhang, L., "Why TCP timers don't work well," *Proceedings of ACM SIGCOMM '86,* Stowe, Vermont, pp. 397-405, 1986.

Figure 2. Experiment 1: Observed RTTs vs Packet Numbers. Simulated SRTTs and RXTs vs Packet Numbers, assuming 500ms clock resolution in RTT values.

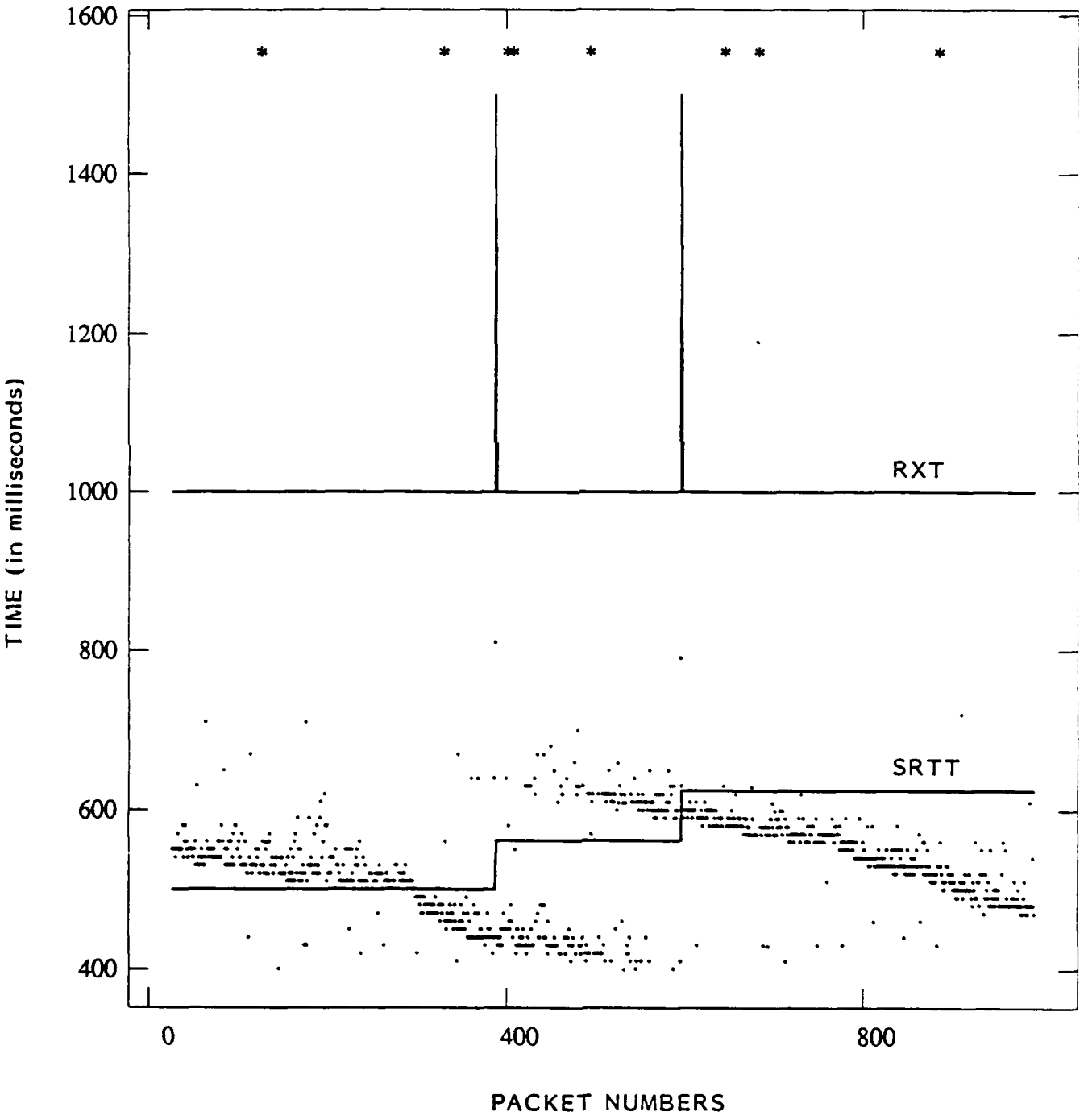


Figure 3. Experiment 1: Observed RTTs vs Packet Numbers. Simulated SRTTs and RXTs vs Packet Numbers, assuming 10ms clock resolution in RTT values.

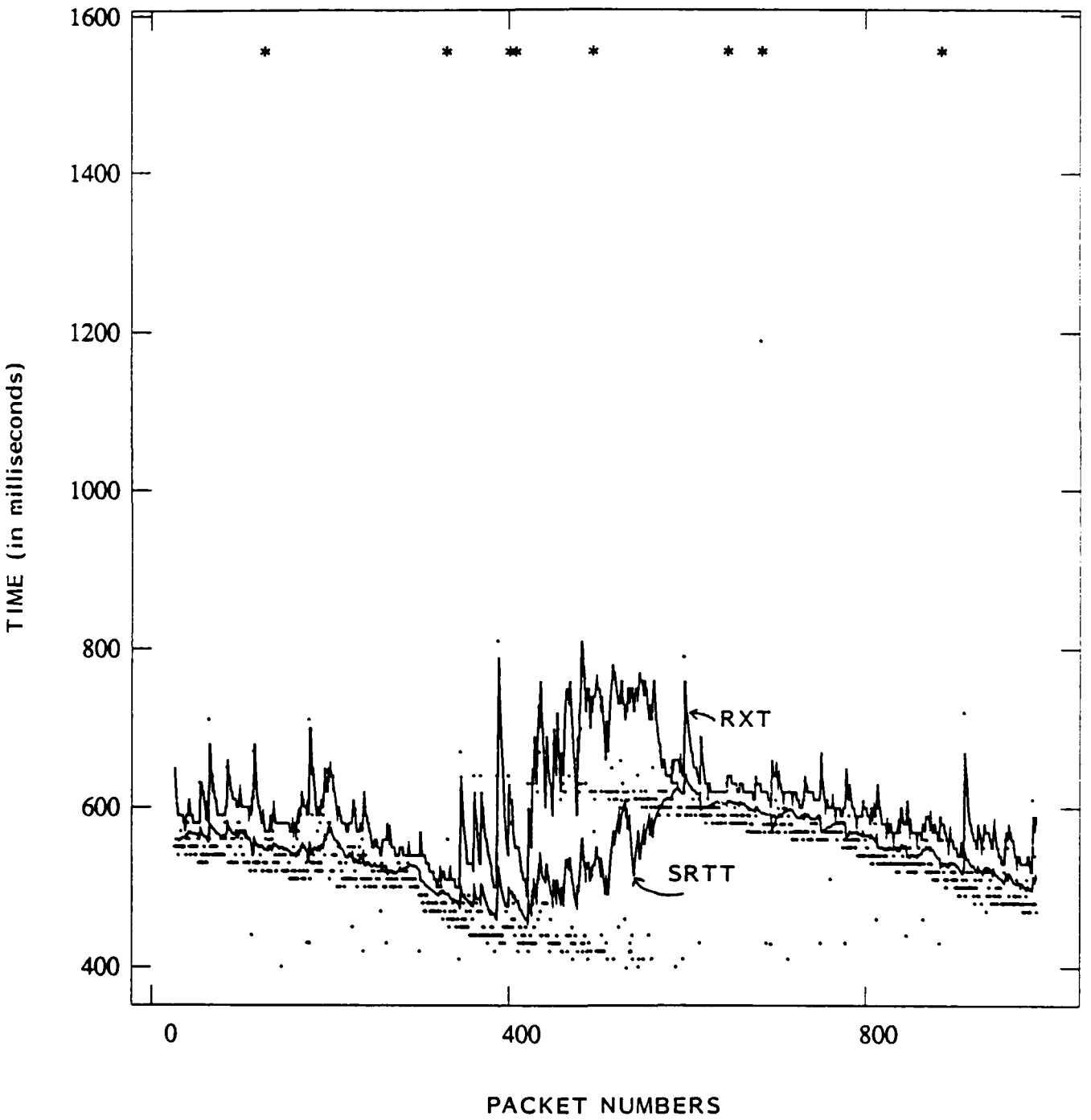


Figure 4. Experiment 2: Observed RTTs vs Packet Numbers. Simulated SRTTs and RXTs vs Packet Numbers, assuming 500ms clock resolution in RTT values.

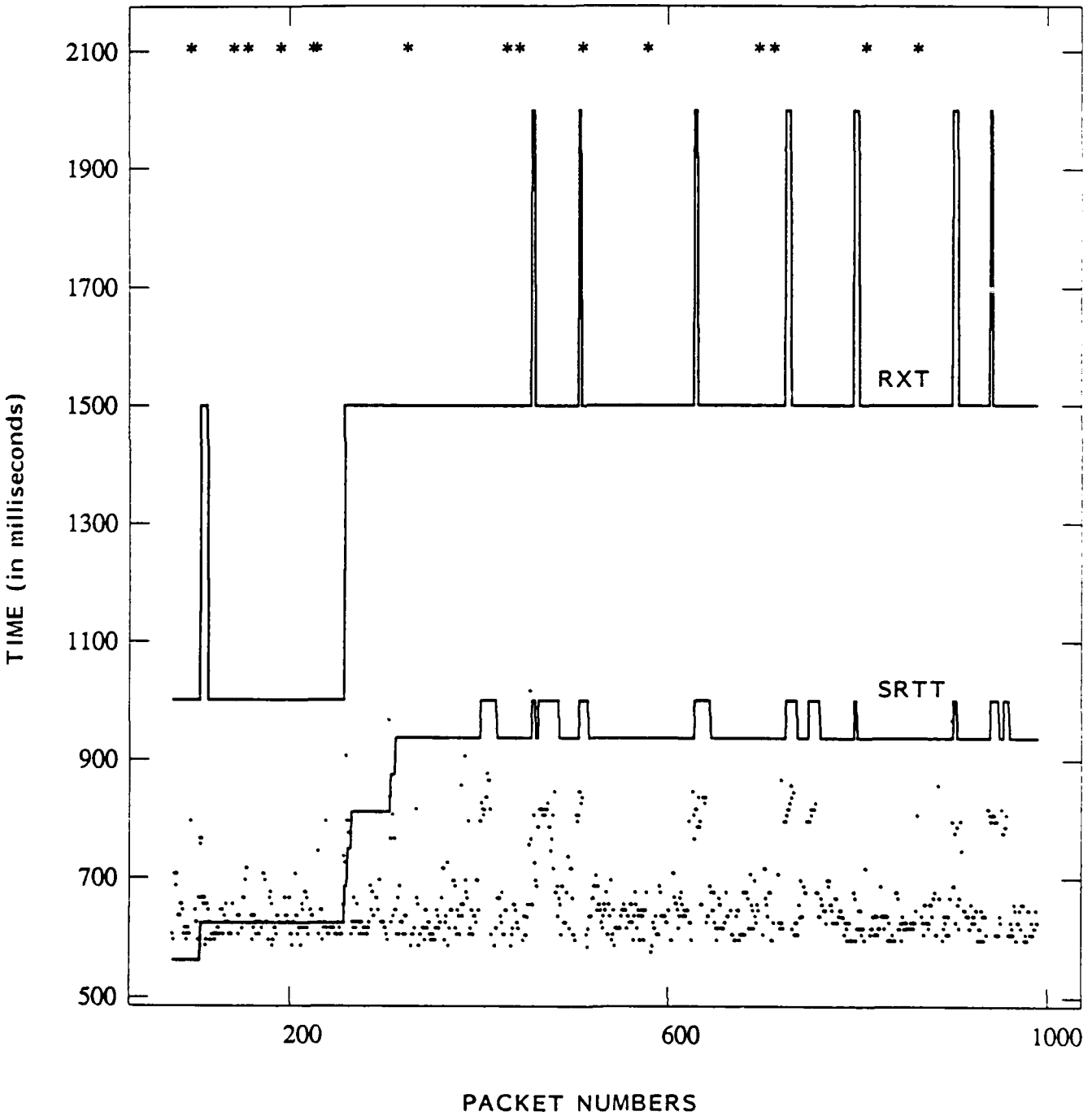


Figure 5. Experiment 2: Observed RTTs vs Packet Numbers. Simulated SRTTs and RXTs vs Packet Numbers, assuming 10ms clock resolution in RTT values.

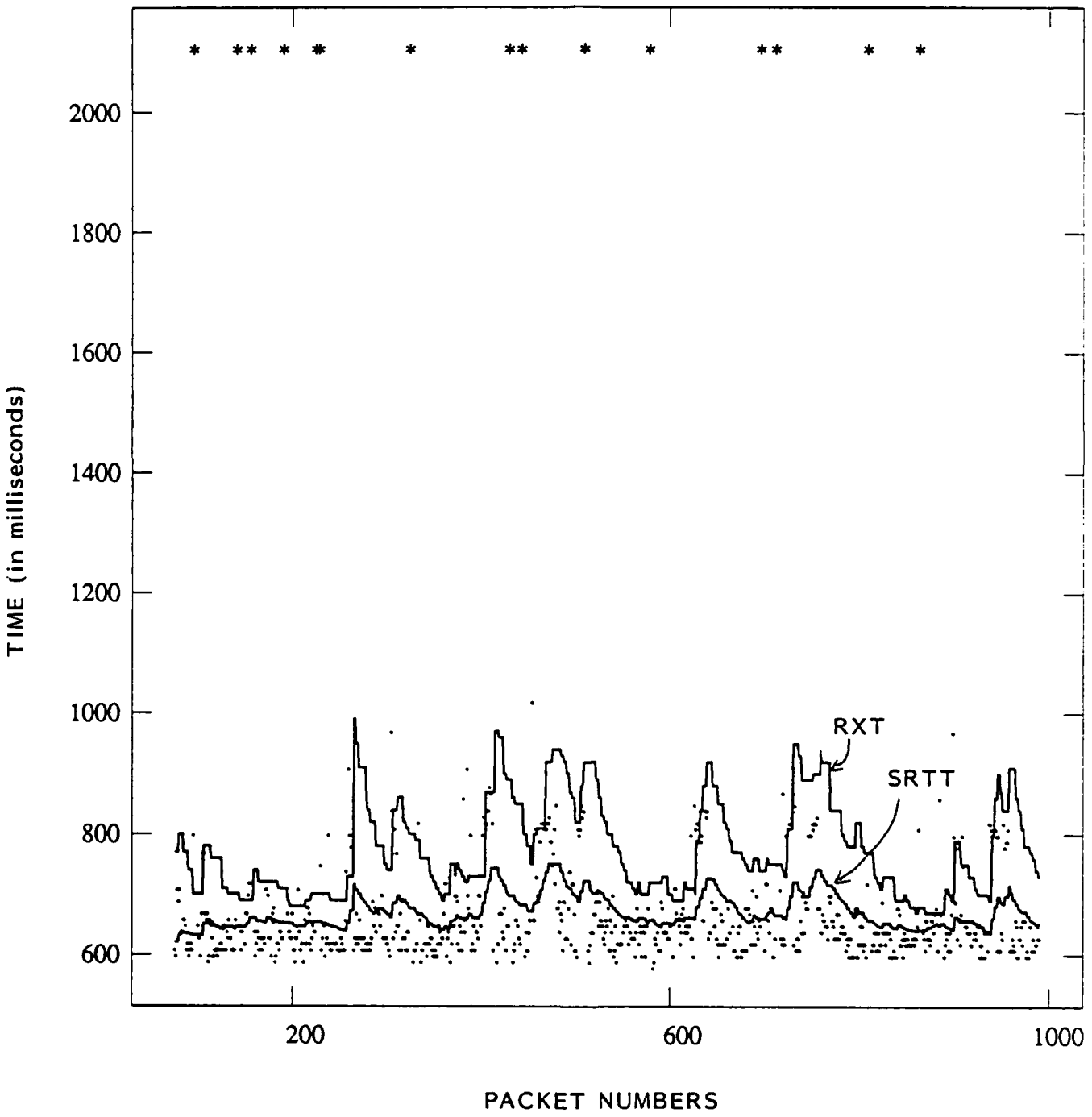


Figure 6. Experiment 1: Error metrics vs α , with $\alpha = \alpha'$

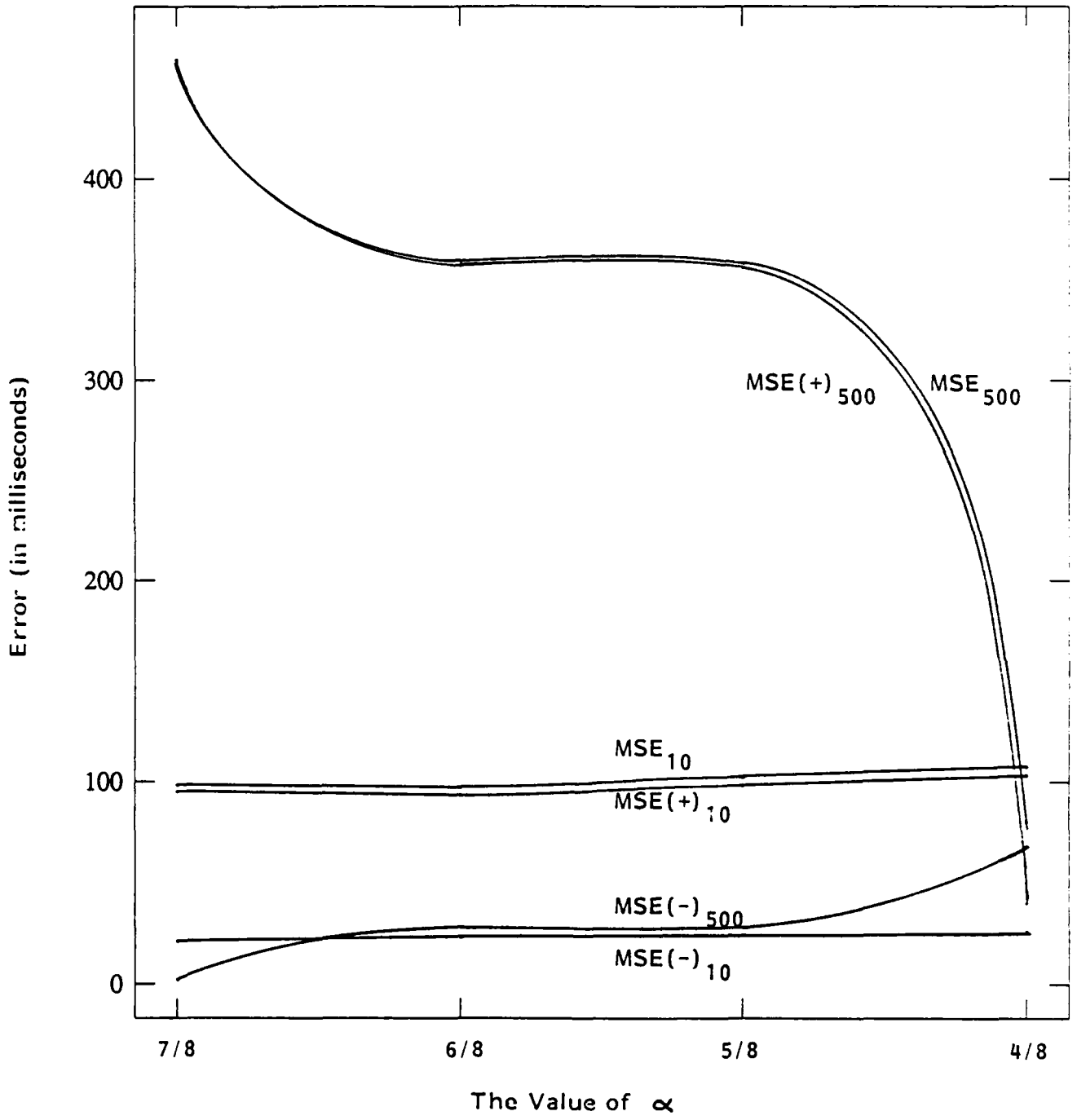


Figure 7. Experiment 2: Error metrics vs α , with $\alpha = \alpha'$

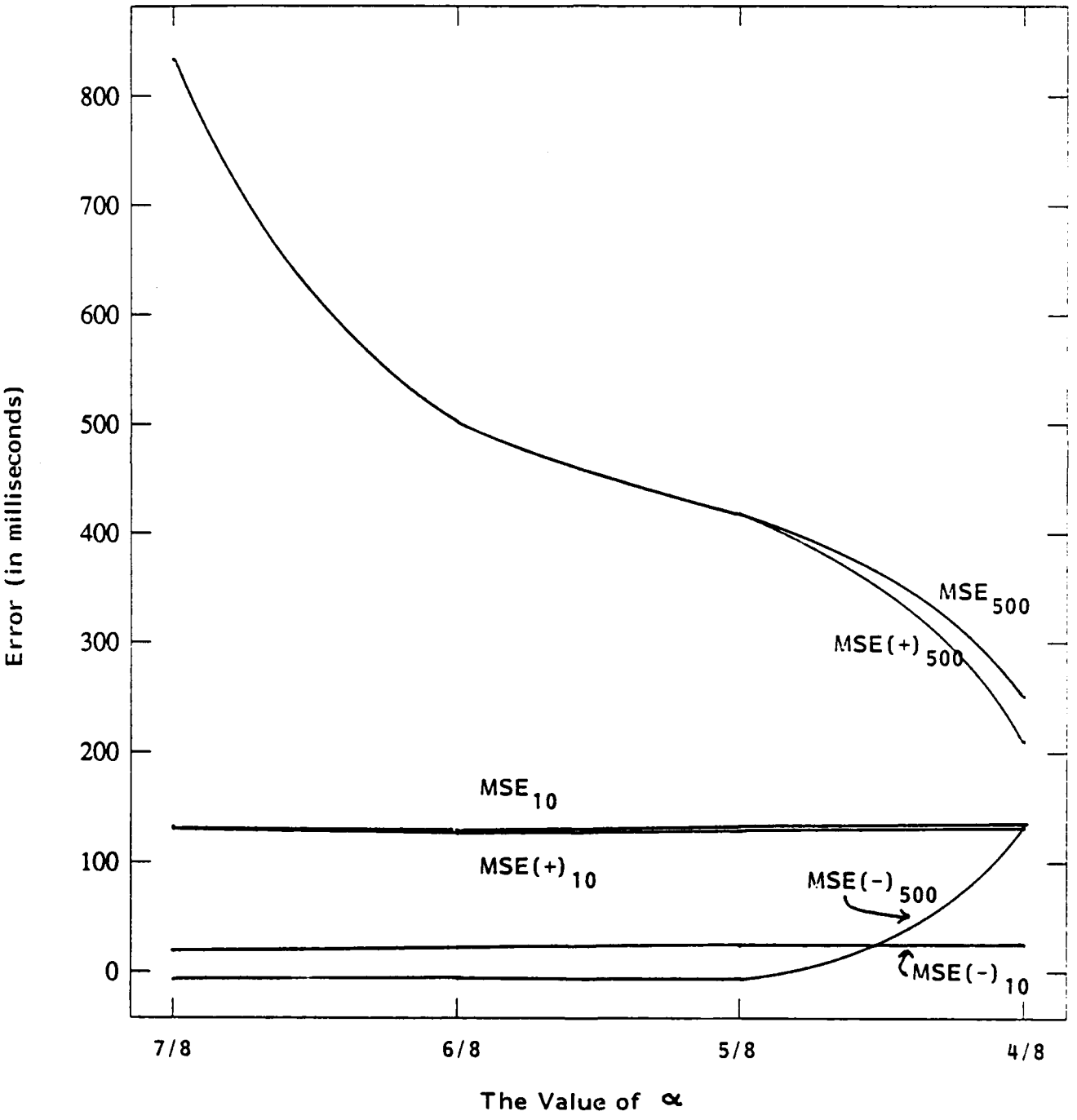


Figure 8. Experiment 1 with simulated SRTTs and RXTs vs Packet Numbers, using RTTs of all possible packets.

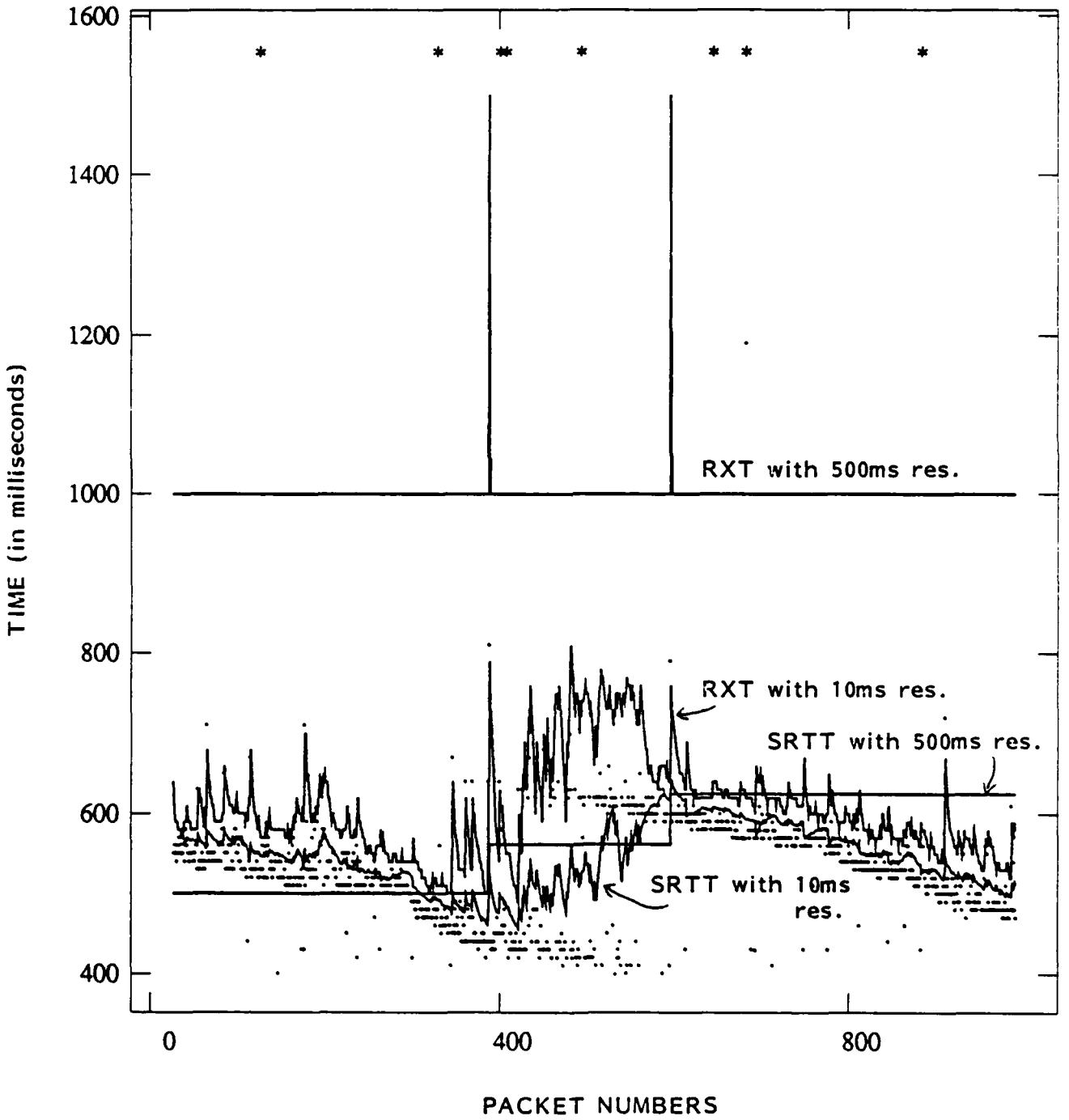


Figure 9. Experiment 2 with simulated SRTTs and RXTs vs Packet Numbers, using RTTs of all possible packets, assuming 500ms clock resolution in RTT values.

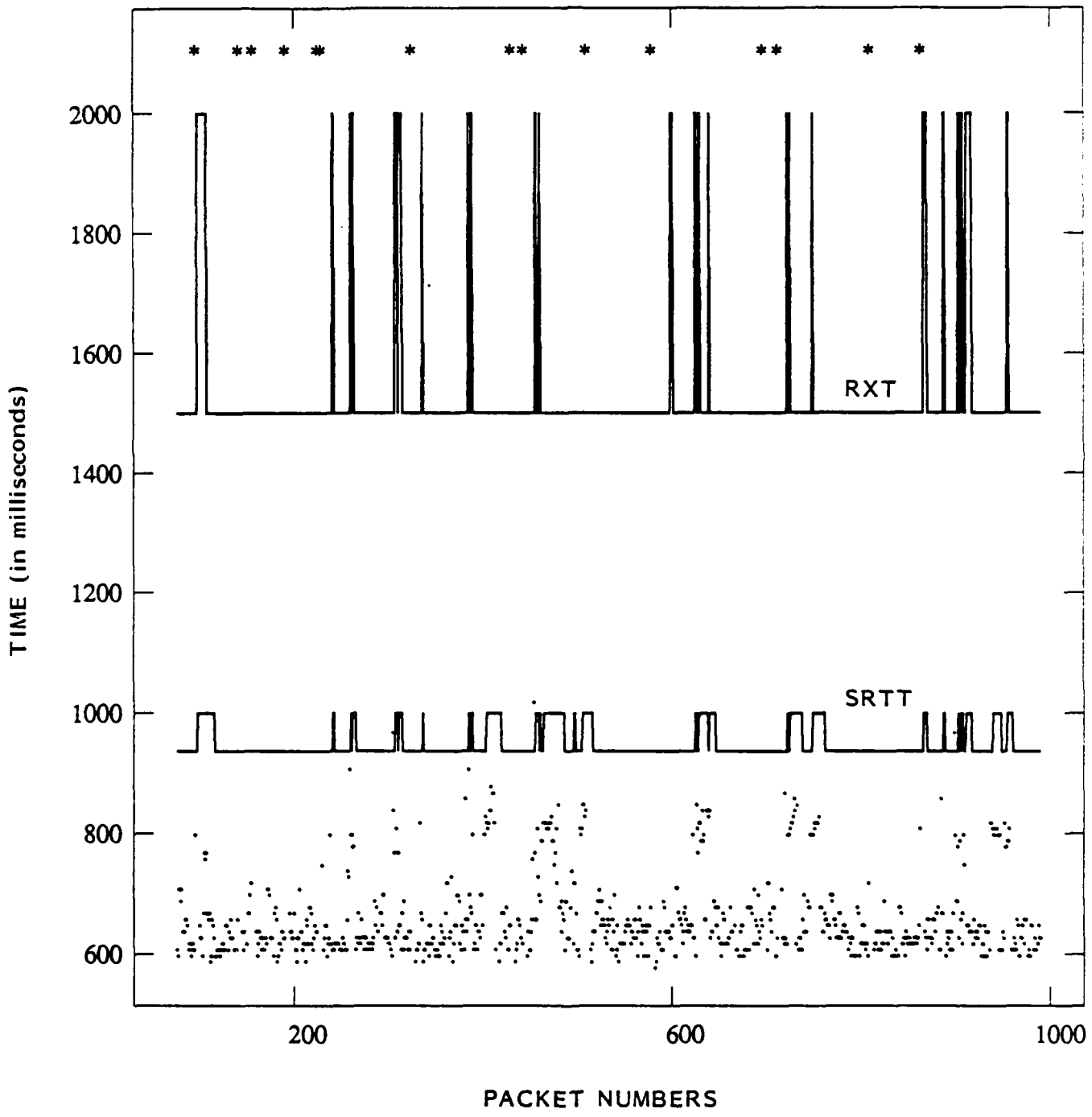
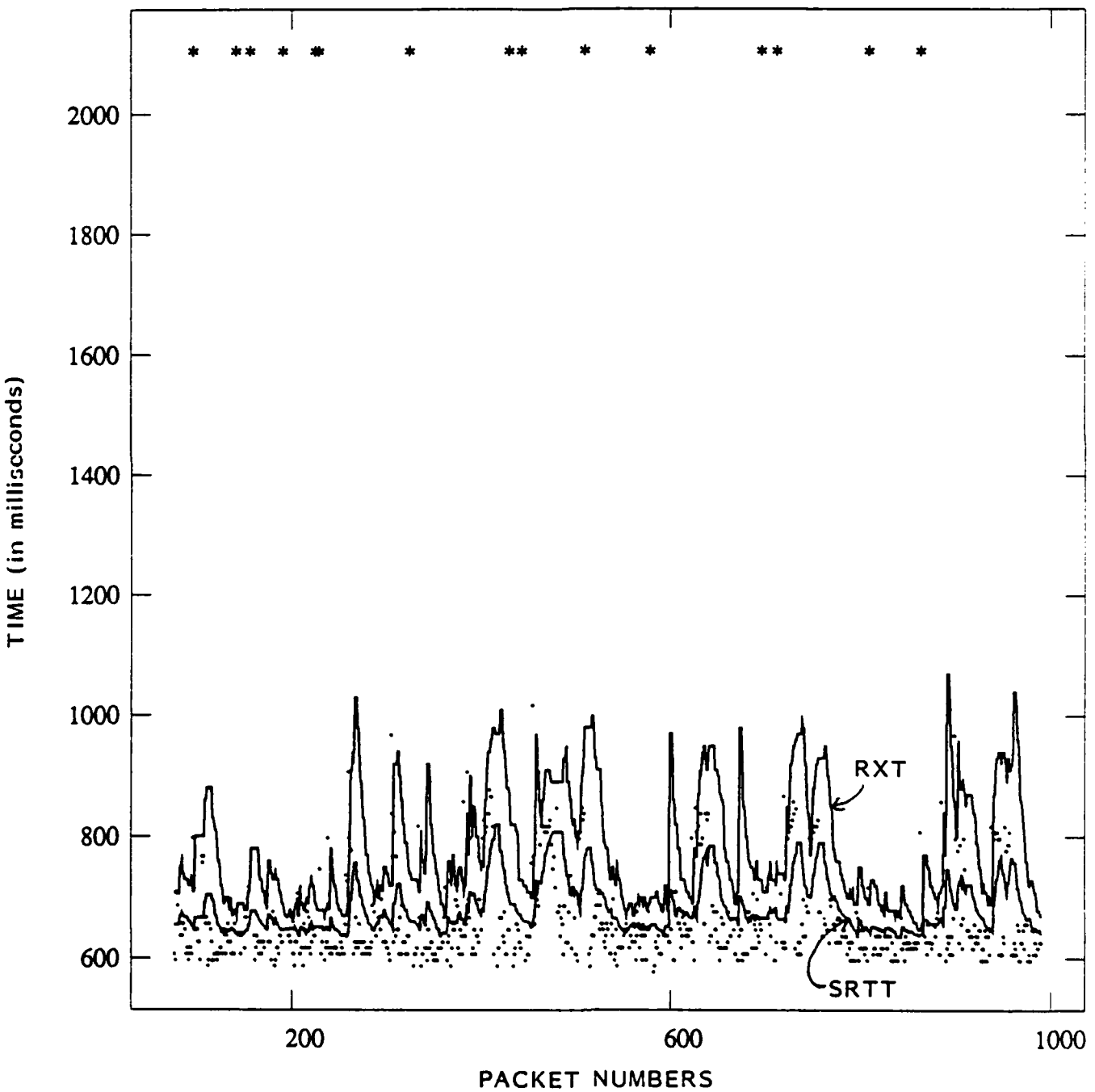


Figure 10. Experiment 2 with simulated SRTTs and RXTs vs Packet Numbers, using RTTs of all possible packets, assuming 10ms clock resolution in RTT values.



UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS N/A
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UMIACS-TR- 90-38 CS-TR- 2432	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION University of Maryland 6b OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION U.S. Army Strategic Defense Command / Office of Naval Research
6c. ADDRESS (City, State, and ZIP Code) Department of Computer Science University of Maryland College Park, MD 20742	7b. ADDRESS (City, State, and ZIP Code) Contr & Acq Mgt Ofc. CSSD-H-CRS, P.O. Box 1500 Huntsville, AL 35807-3801 800 N. Quincy Str. Arlington, Va 22217-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER DASG60-87-C-0066 / N00014-87-0241
8c. ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)

A TCP Instrumentation and Its Use in Evaluating Roundtrip-time Estimators

12. PERSONAL AUTHOR(S)
D. Sanghi, M.C.V. Subramaniam, A. Shankar, O. Gudmundsson, P. Jalote

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical Report	FROM _____ TO _____	1990, March	31

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

We describe an instrumentation of TCP/IP that monitors TCP connections and provides values of internal variables of the implementation. We define interface events for a TCP/IP connection, describe how traces are obtained, and how application processes initiate trace collection. The instrumentation has been implemented in 4.3BSD UNIX. The instrumented TCP/IP provides a flexible environment for experimental studies. Using the instrumentation, we have studied the performance of different roundtrip-time estimators in the Internet environment. One conclusion of our study is that clock resolution is an important parameter, and the resolution currently used in UNIX implementations of TCP is woefully inadequate. Another conclusion is that, with an adequate clock resolution, a recently proposed estimator performs substantially better than the estimator suggested in the TCP specifications.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) (301) 454-4968 / 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED